Explicit Assumptions enrich Architectural Models

Lago, P.; van Vliet, H.

published in
Proceedings 27th International Conference on Software Engineering (ICSE 2005)
2005

document version
Publisher's PDF, also known as Version of record

Explicit Assumptions Enrich Architectural Models

Patricia Lago
patricia@cs.vu.nl

Hans van Vliet
hans@cs.vu.nl

Department of Computer Science
Vrije Universiteit Amsterdam
The Netherlands

ABSTRACT

Design for change is a well-known adagium in software engineering. We separate concerns, employ well-designed interfaces, and the like to ease evolution of the systems we build. We model and build in changeability through parameterization and variability points (as in product lines). These all concern places where we explicitly consider variability in our systems. We conjecture that it is helpful to also think of and explicitly model *invariability*, things in our systems and their environment that we assume will *not* change. We give examples from the literature and our own experience to illustrate how evolution can be seriously hampered because of tacit assumptions made. In particular, we show how we can explicitly model assumptions in an existing product family. From this, we derive a metamodel to document assumptions. Finally, we show how this type of modeling adds to our understanding of the architecture and the decisions that led to it.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Documentation

Keywords

Software architecture, architecture model, assumption, documentation, knowledge management

1. INTRODUCTION

From a software engineering perspective, software that does not change, is not very interesting. We need not bother about the code's comprehensibility, because no one has to read it any more. The quality of the architecture is not important, because no one will change it. The complexity of the system is irrelevant, because it bothers no one.

Software is only of interest when it does change. Then issues like comprehensibility, quality, complexity, suddenly do become important. Conversely, when developing systems, we should not only bother about today's requirements but also, and maybe even more so, about the requirements of tomorrow.

This is one of the main reasons for the importance of software architecture, as for instance stated in [1]: a software architecture manifests the early design decisions. These early decisions determine the system's development, deployment, and evolution. It is the earliest point at which these decisions can be assessed.

There are many definitions of architecture. Many talk about components and connectors, or the 'high-level conception of a system'. This high-level conception then is supposed to capture the 'major design decisions' [1]. Whether this is really the case can only be ascertained after the fact, when we try to change the system. Only then it will show which design decisions were really important. A priori, it is often not at all clear if and why one design decision is more important than another [8].

Since the quality of a system is to a large extent determined by the changes we can still make to that system we, as designers and architects, go to great lengths to try to predict the future user requirements. Any change we foresee, we build into the system, through a proper decomposition, the right kind of interfaces, parameterization, and so on. In software product lines, we explicitly model variability: places where we expect versions of the system to vary in some respect [2]. In software architecture assessments, we consider anticipated changes to a system to determine its long-term robustness.

By doing so, we make assumptions about what will change and, mutatis mutandis, about what will *not* change. At a later stage, we may get into trouble if something changes that we had not foreseen. Then, maintenance becomes expensive, the next release takes too much time, or the architecture is found to be too inflexible.

We conjecture that it is not only useful to explicitly model variability at software architecture time, but that it is also useful to explicitly model *invariability*, i.e. the assumptions we make about the system and the environment in which it is going to function. In a sense, we argue for explicitly modeling "negative" as well as "positive" variability, where negative variability concerns things that we assume will not change.

We see three important uses for the explicit modeling of assumptions:

- **Traceability**. We may trace assumptions to design and implementation solutions, and vice versa [21]. Like other traceability information, this is of help during both development and evolution. In particular, it may help prevent complex change processes that go against major assumptions made at architecture time.
- **Assessment**. Software architecture assessments often make use of change cases [4], use cases that describe potential fu-

ICSE'05, May 15–21, 2005, St. Louis, Missouri, USA.

ture requirements. These change cases are then used to assess the system's modifiability. In a similar vain, we may ask ourselves what will happen if a certain assumption proves to be invalid.

- **Knowledge management**. We may explicitly model an architecture and its associated assumptions, in the same way as we model an architecture and its variants [10], or a design space of possible architectural solutions [3]. This externalizes architectural knowledge present in an organization, and is the basis for later reuse.

This is an exploratory study. We give examples from the literature, and report on experiences we have had with evolving systems to corroborate the idea that the explicit modeling of assumptions is worthwhile. In particular, we explicitly model assumptions in the architectural views of an existing product family, namely the product feature models and product component models.

Based on this, we derive a metamodel that provides the required formal basis for documenting assumptions and their relations with architectural assets. Our study shows that this type of modeling adds to our understanding of the architecture and the decisions that led to it. In particular, we illustrate each of the aforementioned uses of explicitly modeled assumptions.

The remainder of this paper is organized as follows. In sections 2 and 3 we elaborate on why assumptions are important, and the kinds of assumptions one may think of. Section 4 gives some smaller examples. Section 5 discusses a larger example of a product and some of the assumptions behind it, including the incorporation of these assumptions in architectural models and what can be learned from doing so. Section 6 defines the meta model representing assumptions and the relations with architectural assets. Section 7 discusses related work, and section 8 gives our conclusions.

2. WHY MANAGE ASSUMPTIONS?

The systems we build function in a certain environment. In a very general sense, this environment consists of technology, organization, and management. The technology concerns hardware and software: operating systems, programming languages, middleware platforms, display technology, network infrastructure, and the like. The organization consists of business functions and operating procedures. Management concerns the different levels of management and their roles and information needs.

Changes to the systems we build are caused by changes in their environment [16, 15]. Some of these changes have been anticipated during the design; these have been catered for using any of the mechanisms mentioned before: from separation of concerns to limit ripple effects of changes to variation points for the handling of variability. We might term these explicit assumptions. The corresponding changes can be accommodated with relative ease. It is the changes *not* catered for in the architecture that bring us trouble. We might term these implicit assumptions.

Of course, we cannot cater for, or foresee, all possible changes. There's a limit as to how far we can predict the future. Time and money available further limit our possibilities, both with respect to the requirements engineering effort and the actual system to be implemented. But it helps to be conscious not only about the future evolutionary capabilities of the anticipated system, the things that may change, but also about things that had better not change. If these invariabilities can be made explicit, we can reason about them and make conscious decisions about the system's inflexibility with respect to those assumed invariabilities. If they are not made explicit, or if we do not consider them at all, chances are that we get into trouble during the evolution of the system. In particular, this is likely to happen if these invariabilities are reflected in the girders of the system's architecture.

3. REQUIREMENTS, CONSTRAINTS, ASSUMPTIONS

It is quite tricky to draw the line between assumptions, requirements and constraints. We often use these terms interchangeably, and we can always find some example that contradicts a definition. But we may argue as follows.

Requirements are demands on a system, and can be both functional and non-functional.

Constraints are limitations on a system. We tend to think of constraints as non-functional requirements (like the use of a certain database technology, the implementation in a pre-defined programming language), but they be functional too (e.g. integration with the functionality of an existing sub-system).

Requirements and constraints are always conscious desiderata on the system. They are always stated by the client or user of the system, and therefore they must be explicit (otherwise we will never know them!).

Assumptions are the reasons for architectural decisions that are more or less arbitrarily taken on the fly because of personal experience, background, domain knowledge, the artifacts reused, and the like. They are made by the developer or architect of the system and they are never explicit upfront. They are made implicitly and can, but need not, be made explicit on the way.

When left implicit, assumptions are much more difficult because they remain tacit. They are more interesting as well, as they define the hidden long-term invariabilities the system has to adhere to beyond requirements and constraints.

Following the general information systems literatures (e.g. [14]), we may classify the assumptions with respect to their source:

- Technical assumptions. These concern the technical environment in which a system is going to run: programming languages, database systems, operating systems and middleware software are examples.

- Organizational assumptions. These reflect the company as a whole, its social settings and principles. They concern the organizational aspects in the company implicitly brought into software development. Examples include work-flow, organizational structure reflected in development teams and departments, technology adopted as a company standard. Organizational assumptions can refer to the organization developing the software product, or the one using it.

- Managerial assumptions. These reflect the decisions taken to achieve business objectives. They concern the solutions and the operational tasks to achieve organization-level objectives. Examples include management strategies and plans, experience brought into projects, expansions toward new/different market segments.

Earlier work investigating the relation between assumptions and software architecture mainly considers technical assumptions at the component level [9]. This resulted in the notion of architectural mismatch. We rather look into assumptions from a broader perspective: we consider both non-technical assumptions and assumptions that result in cross-cutting issues.

4. ANECDOTAL EVIDENCE

We do not know of examples where invariabilities were explicitly modeled up front and conscious decisions were made based on these assumptions. We do know of many examples where variabilities were explicitly modeled, most notably as feature trees in product lines [2]. Below, we give a number of examples of actual systems and their evolution, and illustrate how some of the difficulties in the evolution of those systems can be traced back to assumptions made:

- The first example concerns a large business information system developed on behalf of Dutch Customs. In 1999, we were asked to assess its architecture. We were especially interested in assessing the capabilities of the system to accommodate future changes. We asked stakeholders to bring forward possible changes to the system, and next investigated how these changes would affect the software architecture. Two years later, we studied the actual modifications to the system. We predicted a lot of changes, but also missed one that severely impacted the architecture [12]. This change concerned the number of staff members that could process a declaration. The original requirements stated that a declaration was to be processed by one staff member, because of privacy rules. Once the system was implemented, the need arose to have large declarations processed by several staff members in parallel to speed up the processing. This change affected almost all components of the system. Arguably, this was a requirement, but the associated performance assumption was not made explicit or checked.
- In [6], the evolution of GRUMPS, a generic middleware, is discussed. The main requirement of version 1 was to be able to communicate remotely. RMI was therefore chosen. Once implemented, there arose the wish to be able to use the system across the Internet. RMI isn't very good at that, since it doesn't work well within the context of firewalls. So a different solution using sockets was chosen in successive versions, to overcome the tacit assumption made earlier.
- One big administrative system in the social security sector assumed a divisionalized organization, where each district had its own database, and its own workflow. This was partly due to the technical possibilities of the era in which this system was developed, and partly because of the relative autonomy of districts. Access from anywhere in the country, to *all of* the information in the system is complex, and if people move from one district to another, manual steps are needed to move their data as well.
- Many commercial software applications need a valid license key to be able to start. This license key is often managed by a license server providing a check interface to clients. This implies that: to be able to use the commercial application, the computer hosting it must be connected to a network, on which the license server is accessible. Network accessibility is an assumption about the environment in which the application executes. If end users use these applications via desktop computers connected to a LAN, this is not an issue; if they use notebooks from different locations (e.g. by traveling), the network connection requirement is a superfluous limitation.

5. A LARGER EXAMPLE

At this point, we may ask "What next? How can we make assumptions explicit? And how should we relate them to current software architecture documentation?". To answer these questions, we carried out an experiment[1], and we are able to make some interesting observations.

This experiment took as input the architectural views of an existing software product in a product family (i.e. the feature models), together with the list of implicit assumptions we identified for the product family. The product family implements end user communication on the integrated Internet-Telephony network. We elaborated the assumptions, and modeled them together with their dependencies on the product features. Below, we discuss three of the assumptions in greater depth, and model them together with (parts of) the feature and structural representation of one of the products. Based on this modeling exercise, we developed a metamodel for documenting assumptions; see section 6. A more elaborate discussion of the assumptions is given in [11].

5.1 Characterization of assumptions

In our experiment, we identified seven different assumptions [11]. Figure 1 shows the feature model that was developed for one of the software products. Features are user-visible aspects or characteristics of a system [2], like functionalities or technologies. A feature model represents a high level view identifying the commonalities and variabilities within a system. For presentation purposes, the model has been simplified by hiding the features not directly influenced by the first two assumptions discussed below.

Features are depicted as rectangles in figure 1. Features added because of the modeling of assumptions are in gray. Assumptions are depicted as a rounded box, and include a label `Assumption`. Because of limitations of the tool used, external features (e.g. COTS) are also depicted as rounded rectangles.

Assumption 1 concerns the completeness of base technology; see Figure 2. The explicit representation of this assumption in the product feature model is found at the right side of Figure 1. To model this assumption, we took the following steps:

1. **Identify the set of features that are directly influenced by the assumption.** In our example, the assumption is that the execution environment is completely specified. This is true if the following features are all properly configured: the programming environment (JVM), the distributed communication platform (CORBA Platform), and the communication technology on both Internet (Our SIP Server) and telephony (Cisco 3640 Router) networks.

   The first two features were completely missing in our original feature model. In the environment where the system was developed, all computers had a standard configuration including JVM and CORBA. To complete the picture, we had to add these features, and analyze their dependencies with the existing features. Also, we had to add the feature `Distributed communication` that classifies them as a type of `Communication Technology`.

2. **Define the dependencies between the assumption and the feature model.** We defined two types of associations to model these dependencies:

   - Association `F-Impacts` defines the dependency between the assumption and the set of generic features directly involved. In this case, the assumption "Base technology is complete" impacts the four features identified in the previous step. E.g., feature `JVM` must be

[1]The term *experiment* is here used according to the definition given in [5], i.e."*an intentional exercise carried out to gain new knowledge [. . .]*".

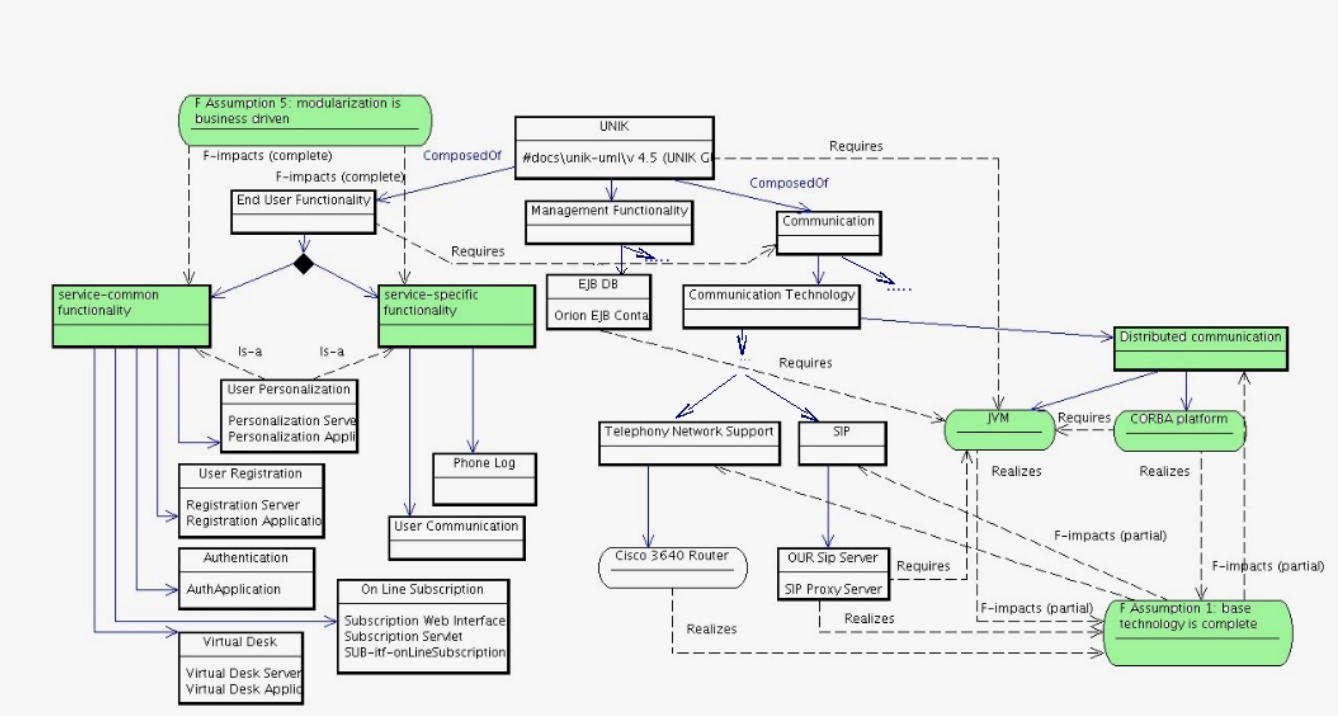

Figure 1: Modeling assumptions in a product feature model

A software product typically relies on a certain configuration of its execution environment. This configuration includes both special purpose technology (e.g. a certain distribution middleware) and general purpose technology (e.g. a certain Java VM version). We usually keep track of the first technology type, while we implicitly assume that the second type of technology is in place. This of course leads to inconsistencies and conflicts when we reuse components originally based on different technologies and hence when we need to reconstruct their original execution environment.

Figure 2: Assumption 1: Tracing base technology versions

properly configured, because the Java VM is needed to execute all Java components. The impact is called partial because `JVM` also functions as a programming and/or execution environment.

Further characteristics pertaining to these impacted features are modeled as feature attributes capturing the specific technical information necessary to restore the specific technology configuration. For example, it is not sufficient to include the JVM in the base technology: we need to also document e.g. the name of its vendor (e.g. Sun) and its version (e.g. J2SE 1.4.2).

- Association `Realizes`, defined from feature to assumption, shows which features are needed to fulfill the assumption. In our example, the base technology is made up of the resources needed to route end user communication on both the Internet (modeled by the concrete feature `Our SIP Server`), the telephony networks (modeled by the concrete feature `Cisco 3640 Router`), and the `JVM` and `CORBA platform`. Again, making the assumption explicit helped us in reconstructing undocumented knowledge.

Assumption 5 concerns the rationale behind the way the system has been modularized into components; see Figure 3. It is modeled on the left side of Figure 1. To be able to represent this assumption, we took the following steps:

1. **Identify the set of features that are directly influenced by the assumption.** This assumption classifies features as `Service-specific Functionality` or `Service--common Functionality`. The first can be used in a particular service only; the second is independent from the specific service, and can be shared among multiple services. For instance, `Authentication` is service-common whereas `Phone Log` is service-specific.

   Also for this assumption, two features have been added to our model, as the classification of features was implicit in our original model.

2. **Define the dependencies between the assumption and the feature model.** Similar to the previous assumption, association `F-Impacts` has been used to model the dependency between the assumption and the set of generic features directly involved. In this case, the impact is complete, because the two features are completely impacted by the assumption. While assumption 1 identifies a pool of four features that are all needed at the same time, this assumption organizes the product features in two categories, and each feature can in principle belong to one, or both. Due to this difference, a new association has been defined between features. Association `Is-a` indicates whether a feature is service-common or service-specific (or both).

In the design of our product, the overall software architecture was organized in components partially reflecting the business role played by the different industrial partners. As partners were geographically dispersed and (most importantly) they played well-established business roles in the market, it was politically important to clearly separate service-specific features from service-common ones: the first provide functionality that is specific to a particular type of application (e.g. video on demand); they are commercialized by service providers. The second provide basic functionality reusable across different applications; they are commercialized by vendors and network providers.
In a similar way, service-specific behavior and communication management are separated. To ensure this separation, additional components were needed to mediate the interaction between the two levels. In both cases, the distribution of functionality on components reflects the business model of the application domain. This can lead to lower quality (e.g. decreased performance) in the final product.

Figure 3: Assumption 5: Business-driven modularization

As a final example, assumption 4 concerns the support of security by all access points. We followed the same process discussed above to identify the set of features influenced, and defined the dependencies between the assumption and the feature model. The result is depicted in the upper half of figure 4. The assumption impacts three features: `End User Functionality`, `Distributed Communication` and `User Authentication`. Feature `End User Functionality` recursively decomposes in a number of smaller features (not shown for reasons of space), each of which has to pay attention to security, i.e., security is a cross-cutting issue. The actual impact of the security assumption can more clearly be shown at the structural level of interfaces and modules, as shown in the bottom half of figure 4. In there, we show the impact of this assumption on the (interfaces of) COTS or external components like `Registration App`, and module `Authentication Server`. This association is labeled `S-impacts` to distinguish it from the `F-impacts` at the feature level.

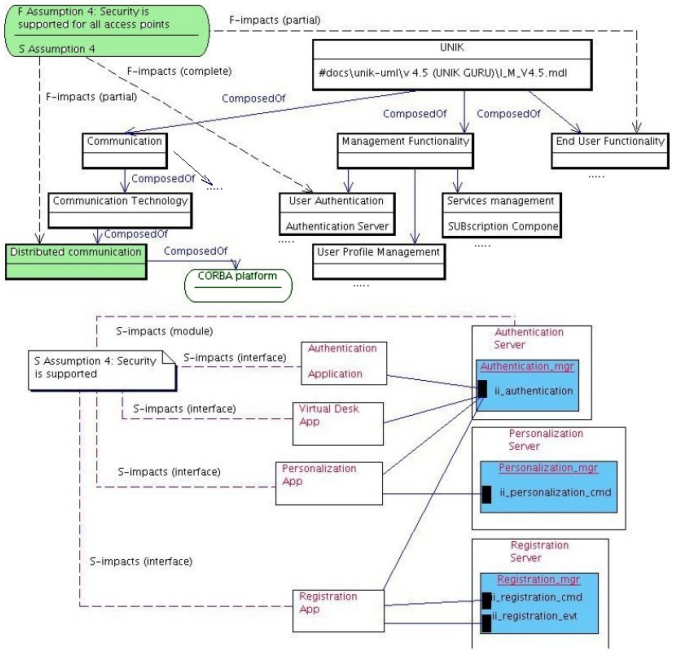

Figure 4: A cross-cutting assumption

5.2 Observations

From this experiment in explicitly modeling assumptions in a product feature model, we can make the following observations:

1. For both assumptions 1 and 5 we had to add new features in order to be able to characterize the assumption in the existing feature model. This demonstrates that documenting assumptions provides further insight in the software system, and helps to transfer the knowledge about implicit, recurring decisions embedded in the solutions chosen.

2. In assumption 5, a feature can in principle be both service-common and service-specific. If this happens, association `Is-a` does not provide enough details about which factors inside a certain feature are service-common and which are service-specific. In this particular case, the reason is that the assumption is structural in nature (the "service level" is decided on structural entities like e.g. components), whereas features are functional entities, and the structural entities implementing them are represented in a different architectural view (e.g. the component model). This indicates that assumptions may belong to different abstraction levels, and that they should be represented in different architectural views.

   For example, feature `User Personalization` (in Figure 1) is implemented by two components (also listed in the lower part of the feature box; represented as links to the component model) Personalization Server and Personalization Application, having different owners. Let us suppose that the first be service-common and the latter service-specific. In this case, we cannot model the difference in this view, as the `Is-a` association is defined between features. What we can show, is that feature `End User Personalization` is related by association `Is-a` to both features `Service--common Functionality` and `Service-specific-Functionality`.

   To make this difference explicit, we need to represent the assumption and/or its features in the component model too (as in Figure 5), and document which features are provided by which structural elements.

3. In assumption 1 it is interesting to notice that there is a kind of dual dependency between features and assumptions: the `F-Impacts` association from assumption to feature shows the most generic features (in the feature decomposition tree) that are influenced by the assumption. The other way around, the `Realizes` association from feature to assumption indicates which specific implementation of that feature is actually involved in that assumption.

   If generally applicable, this duality can be used to identify the impact level of a certain assumption, by analyzing how far in the feature decomposition tree it comes.

4. We may say that an assumption is *cross-cutting* if it affects more than one feature or structural element. This is directly covered by the impact associations in our model. This in itself is not new.

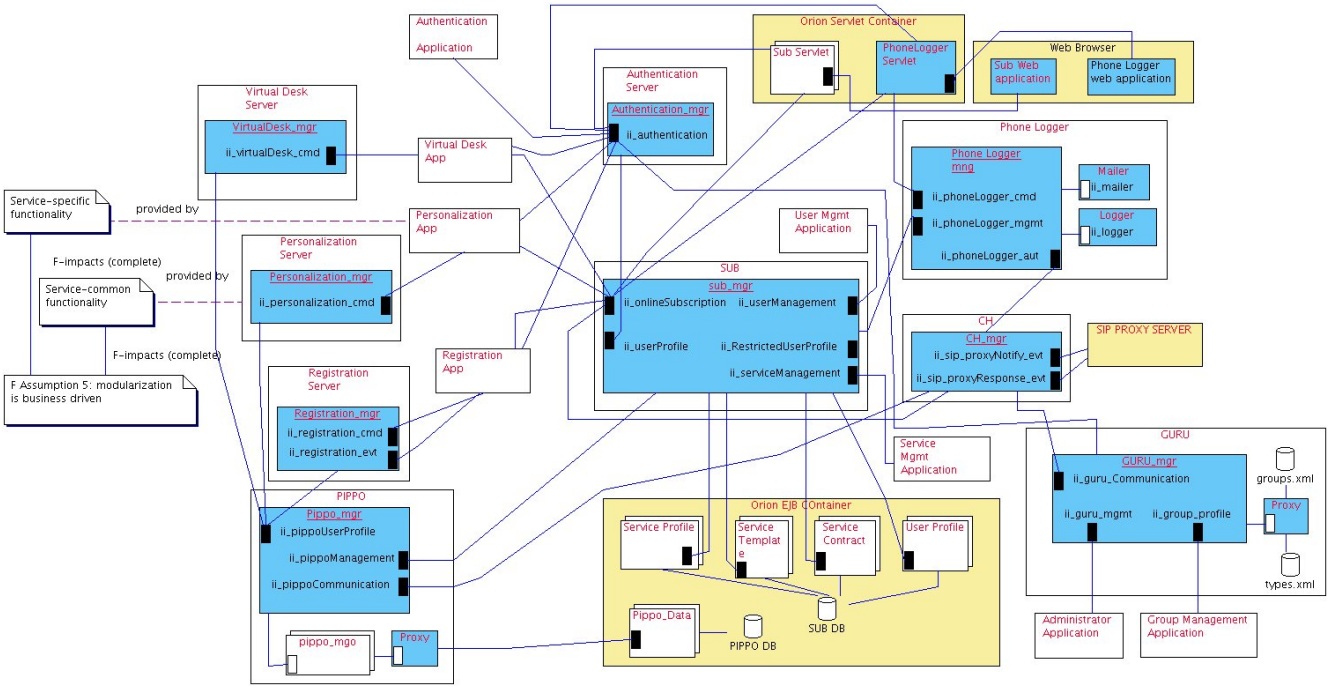

Figure 5: Modeling assumptions in a product component model

By rephrasing the definition of "cross-cutting aspect" given in [18] we can say that an assumption is cross-cutting in a target structure (e.g., a feature tree or a component model) if information about the assumption is mixed with information about other assumptions. This is another level of cross-cutting. According to this definition, we can identify cross-cuts at both feature and structural levels. For example, at the feature level both assumptions 4 and 5 impact (among others) the `End User Functionality` feature; this means that information about these assumptions is mixed in the feature. At the structural level, we can see in Figure 4 that assumption 4 impacts both the external interfaces of end-user applications and the complete component implementing the Authentication server. Assumption 5 at the structural level (see Figure 5) involves only one end-user application, the Personalization Application, which provides the needed service-specific functionality.

Our notation not only allows to identify this type of cross-cutting issue at both levels, but also helps to precisely locate the cross-cutting at the structural level. This is an important issue that is rarely addressed in the literature, which usually focuses on modeling assumptions between components, or between a component and its execution context.

5. When modeling some assumptions, like assumption 5, we had to extend the feature model with the new association `Is-a` that further classifies features and that refines the existing domain knowledge.

6. Whenever we introduce new features we need to find their location in the feature model. Their parent feature in the decomposition tree needs to be identified, and if this is not already modeled we may have to apply changes to the way features are classified. For example, by modeling assumption 5 we added two new generic features, `service-common functionality` and `service-specific functionality`. These introduced the classification of end user functionality in these two subtypes. An end user feature can in principle be one or the other, and this change could be accommodated with rather easily by simply adding one level in the subtree rooted by the `End User Functionality` feature. If also the `Management Functionality` and / or `Communication` features could be service-specific, then this could have jeopardized how the domain features were classified, and hence the complete feature tree.

5.3 Using the assumptions explicitly modeled

We mentioned three important uses for the explicit modeling of assumptions: traceability, assessment, and knowledge management. Each of these (overlapping) uses can be illustrated with the example from section 5.1. Each shows that the explicit representation of assumptions provides valuable additional information.

Modeling assumption 1, the completeness of base technology, made us realize the dependency between the programming environment and the SIP server, a COTS component. Changing the Java platform may thus incur costs not foreseen, or may not be feasible at all if that version of the COTS component is not available.

Assumption 5 necessitates that service-common and service-specific elements be kept separate, to ensure that e.g. outsourced components are well-defined. This may lower the performance of the service, but enhances maintainability. When this assumption is not made explicit, performance considerations may later on make us oversee this restriction and violate it to increase speed.

While assessing the architecture for modifiability, assumption 1 explicitly identifies the dependencies with the underlying technology, and thus identifies architectural elements that need adaptation if this technology changes.

Finally, when planning to reuse, say, the Telephony Network Support component, our representation highlights its dependence

on certain base technology, which is relevant when assessing the usability of this component in some other environment.

6. ASSUMPTIONS META MODEL

Our assumption metamodel is depicted in figure 6. It is based on the examples discussed above, as well as the modeling of the other assumptions discussed in [11]. Not all of the elements in the metamodel are therefore visible in the examples given above.

The central notion is that of an assumption which impacts a feature. In the feature tree, the leaf nodes represent concrete features, and the higher-level nodes represent abstract features. A given assumption impacts one or more concrete features. The actual association is then made with the root of the smallest subtree all of whose concrete features are impacted. So we associate an assumption with the "smallest" abstract feature that covers all concrete features impacted.

The impact association is either complete or partial. A complete impact association means that the feature implements the corresponding assumption, and nothing more. A partial impact association means that the feature implements additional aspects as well. For instance, feature `End User Functionality` in figure 4 implements secure access. Assumption 4 impacts this feature, but only partially, since the feature implements the actual end user functionality as well. On the other hand, feature `User Authentication` is completely impacted by this assumption.

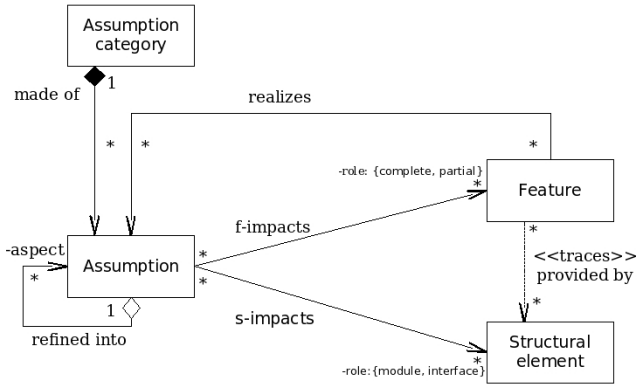

Figure 6: Assumptions metamodel

The association `realizes` links the leaf nodes of the feature tree that are being impacted by a given assumption. The reason for not having just bidirectional links between assumptions and features is that the links are used for different purposes in either direction. Starting at the assumption, and with reuse in mind, we want to know which features are potentially affected by that assumption. This leads to the root of the smallest subtree encompassing them all. Conversely, starting from a concrete feature, and with evolution in mind, the realization link tells us on which assumption this feature depends.

Features are ultimately realized by structural elements such as packages, interfaces and modules. An assumption that impacts certain features, obviously also affects the structural elements that realize that feature. This relation is not always trivial, since there is an N to N relationship between features and the structural elements that realize them. For that reason, we want to explicitly connect an assumption to the structural elements affected. The complete/partial roles at the feature level now translate into module/interface.

Note that we need not always model an assumption at both the feature and structural level. Sometimes, either will do, and sometimes both provide useful information. In the example in the previous section, for instance, assumptions 4 and 5 are modeled at both levels whereas assumption 1 is modeled at the feature level only. Modeling the latter assumption at the structural level would not provide additional information.

Finally, in order to group information on tightly related assumptions, we have introduced categories of assumptions and aspects of assumptions. The first allows us to group similar assumptions under one general umbrella, for example a category "suitable graphical user interface available" with elements "registration component available" and "communication component available". On the other hand, a given assumption may have different aspects that are impacted by (the same or different) features. For example, the assumption "compatible features have compatible data management" has three aspects: "exchanged data has the same format", "data storage is compatible" and "overlapping data is synchronized". For reasons of space, we are not able to illustrate these further; the interested reader is referred to [11].

7. RELATED WORK

The need for explicit documentation of architectural assumptions is not new. Much work reported in the literature focuses on technical assumptions, by looking at the architectural mismatch problem and the inclusion of the notion of assumption.

Garlan, Allen and Ockerbloom [9] address the impact of implicit assumptions on reuse, and discuss the types of problems encountered when implicit assumptions remain undocumented. Besides technical assumptions mainly concerning the nature of components and connectors, they also discuss the global architectural structure and the construction process. Nevertheless, reuse results in an emphasis on the architectural mismatch problem from the components perspective.

Uchitel and Yankelevich [20] also look at assumptions from the component side, and propose a way to extend architectural description languages to include assumptions.

A slightly different approach is taken in [22], which addresses the problem of estimating the costs of integrating COTS components into existing systems. They look from the component side too, but they give a classification scheme applicable to the overall architecture as well, and use that to measure integrability levels.

A completely different perspective is given in [17], which uses scenario-based analysis for forecasting future requirements about information systems: the devised requirements classification covers both technical and non-technical requirements. This interesting work may serve as a starting point for the elicitation of organizational and managerial assumptions.

The literature reports other examples about how assumptions can be considered in various domains, e.g. to integrate cooperating agents [13], to identify variation points shared in a software product line [19], or for requirements detection out of existing implementations [7]. In general, they all approach the problem of making explicit assumptions at the design level rather than at the architecture level, as individual agents, components or modules are used as basis to identify implicit assumptions. Instead, we look for the global picture, and search for assumptions that have an impact on the overall architecture, or parts of it.

8. CONCLUSION

In this paper we advocate the explicit representation of assumptions. In this way, we grasp the invariability which, together with the representation of variability, provides a more complete coverage of the decisions that lead to the architecture of our software systems.

We give a number of examples of actual systems and their evolution, and of the difficulties in the evolution that can be traced back to assumptions made earlier on. Also, we describe an experiment in which we elicit implicit assumptions and make them explicit in the existing software documentation. This experiment suggests a way to model invariability, and allows us to draw some interesting observations that further augment our knowledge.

When made explicit, assumptions can be used in various ways. They provide guidance for reuse: assumptions are naturally expressed at the global level. Hence, they perfectly fit in the software architecture, and can be used as a tool to identify the requirements we want to find back in a new system, and to select the reusable assets that fulfill them and that are also compatible. To this end, traceability from assumptions to reusable assets is needed.

Also, by representing our implicit assumptions in the software architecture, documentation becomes richer. Knowledge can be transferred in a better way, and this gives better support for evolution and maintenance.

Implicit assumptions are cross-cutting concerns: their impact can be (and usually is) diffuse. Our experiment shows that documenting assumptions in the software architecture helps in revealing these cross-cutting paths.

Clearly, our results are initial, and need further investigation. Currently, we are modeling successive versions of a large data mining application and its underlying assumptions to deepen our insights. Issues that need further investigation include:

- Which are the type and amount of assumptions that we can we realistically maintain? Tool support and some kind of semi-automated knowledge management service are necessary.
- When is it more appropriate to model assumptions in a feature view, and when is it more appropriate to do so in a component view? What does either model tell us about the ramifications of assumptions?
- How can we deduce the impact level of assumptions from their explicit model in an architecture description?
- Can we devise guidelines that help us decide *which* assumptions to model? Paraphrasing [8], it is a priori often not clear if and why one assumption is more important than another. Does the categorization of assumptions into technical, organizational and managerial help us identifying the most "relevant" ones?
- How can we employ the explicit documentation of assumptions during impact analysis? Does a categorization into technical, organizational, managerial benefit a stakeholder-oriented impact analysis?

9. REFERENCES

[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley, second edition, 2003.

[2] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, 2002.

[3] H. de Bruin, H. van Vliet, and Z. Baida. Documenting and Analyzing a Context-Sensitive Design Space. In J. Bosch, M. Gentleman, C. Hofmeister, and J. Kuusela, editors, *Software Architecture: System Design, Development and Maintenance, Proceedings 3rd Working IFIP/IEEE Conference on Software Architecture*, pages 127–141. Kluwer Academic Publishers, 2002.

[4] E. F. Ecklund, Jr, L. M. Delcambre, and M. J. Freiling. Change cases: Use cases that identify future requirements. In *Proceedings of OOPSLA '96*, pages 342–358, New York, NY, 1996. ACM.

[5] A. Enders and D. Rombach. *A Handbook of Software and Systems Engineering – Empirical Observations, Laws and Theories*. The Fraunhofer IESE Series on Software Engineering. Addison Wesley, 2003.

[6] H. Evans, M. Atkinson, M. Brown, J. Cargill, M. Crease, S. Draper, P. Gray, and R. Thomas. The Pervasiveness of Evolution in GRUMPS Software. *Software: Practice and Experience*, 33(2), Feb 2003.

[7] S. Fickas and M. S. Feather. Requirements Monitoring in Distributed Environments. In *Proceedings of the 2nd International Workshop on Services in Distributed and Networked Environments*, pages 93–100. IEEE Computer Society Press, June 1995.

[8] M. Fowler. Who Needs an Architect. *IEEE Software*, 20(5):11–13, 2003.

[9] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, Nov. 1995.

[10] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis Feasibility Study. Technical report, Software Engineering Institute, 1990.

[11] P. Lago and H. van Vliet. Observations from the Recovery of a Software Product Family. In *Proceedings of the Software Product Lines Conference*, Lecture Notes in Computer Science, pages 214–227, Boston, USA, Aug. 2004. Springer Verlag.

[12] N. Lassing, D. Rijsenbrij, and H. van Vliet. How well can we predict changes at architecture design time? *J. Syst. Softw.*, 65(2):141–153, 2003.

[13] J. Latimer and A.-M. Chang. A Model of Structured Discourse for Cooperating Intelligent Agents. In *Proceedings of the Hawaii International Conference on System Sciences*, pages 191–200, Maui, Hawaii, Jan. 1996.

[14] K. Laudon and J. Laudon. *Management Information Systems – Managing the Digital Firm*. Prentice Hall, eight edition, 2004.

[15] M. Lehman. Software Evolution – Cause or Effect. Stevens Memorial Lecture, http://www.cs.mdx.ac.uk/staffpages/mml, 2003.

[16] M. Lehman and L. Belady, editors. *Program Evolution*. Number 27 in APIC Studies in Data Processing. Academic Press, 1985.

[17] E. Lewis. The Use of the SAGA Tool for Gathering Requirements for Future Information Systems. In *International Conference on System Sciences – Decision Support and Knowledge-Based Systems*, volume 2, pages 269–278, Maui, Hawaii, Jan. 1996. IEEE Computer Society Press.

[18] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996.

[19] J. Savolainen and J. Kuusela. Consistency Management of Product Line Requirements. In *International Symposium on Requirements Engineering*, pages 40–47, Toronto, Canada, Aug. 2001. IEEE Computer Society Press.

[20] S. Uchitel and D. Yankelevich. Enhancing Architectural Mismatch Detection with Assumptions. In *In Proc. of the Eng. of Computer Based Systems (ECBS 2000)*, pages 138–147, Edinburgh, Scotland, Apr. 2000.

[21] T. Weiler. Modeling Architectural Variability for Software Product Lines. In J. van Gurp and J. Bosch, editors, *Proceedings of the Workshop on Software Variability Management*, pages 55–63, Gröningen, The Netherlands, Feb. 2003.

[22] D. Yakimovich, J. M. Bieman, and V. R. Basili. Software architecture classification for estimating the cost of COTS integration. In *Proceedings of the International Conference on Software Engineering*, pages 296–302. IEEE Computer Society Press, 1999.